闭包 JavaScript 闭包-JavaScript中文网-JavaScript教程资源分享门户 (javascriptcn.com)

闭包就是能够读取其他函数内部变量的函数，函数没有被释放，整条作用域链上的局部变量都将得到保留。 由于在javascript语言中，只有函数内部的子函数才能读取局部变量，因此可以把闭包简单理解成：定义在一个函数内部的函数；闭包的本质就是在一个函数内部创建另一个函数

闭包有3个特性：

①函数嵌套函数

②函数内部可以引用函数外部的参数和变量

③参数和变量不会被垃圾回收机制回收

闭包的用途：

一个是前面提到的可以读取函数内部的变量，另一个就是让这些变量的值始终保持在内存中

```
闭包：将函数内部和函数外部搭建起来的一个桥梁，使函数外部可以用到函数内部的变量
        优点：使函数外部用到函数内部的变量
        缺点：造成内存泄漏

        垃圾回收：
        1.引用计数
        2.标记清除

function count() {
    var n = 0;
    return function(){
         return ++n;
         }
    }
var c = count();//  AO:{n:undefined-->0-->1-->2-->3-->4}
    console.log(c);
    console.log(c());//1
    console.log(c());//2
    console.log(c());//3
    console.log(c());//4

var c1 = count();//AO:{n:undefined-->0-->1-->2}
    console.log(c1());//1
    console.log(c1());//2
    console.log(c());//5
//在这段代码中，count()中的返回值是一个匿名函数，这个函数在count()作用域内部，所以它可以获取count()
```

```
//作用域下变量n的值，将这个值作为返回值赋给全局作用域下的变量c,实现了在全局变量下获取到局部变量中的变量的值
```

闭包会消耗内存：

```
function fn({
    var num = 3
    return function(){
    var n- 0;
    consoLe.log(++n);
    consoLe. log(++num);
   }
}
var fn1=fn();
fn1()//1 4
fn1()//1 5
//一般情况下，在函数fn执行完后，就应该连同它里面的变量一同被销毁，但是在这个例子中，
//匿名函数作为fn的返回值被赋值给了fn1，这时候相当于fn1=function(){var n = 0 … },
//并且匿名函数内部引用着fn里的变量num，所以变量num无法被销毁，而变量n是每次被调用时新创建的，
//所以每次fn1执行完后它就把属于自己的变量连同自己一起销毁，于是乎最后就剩下孤零零的num，
//于是这里就产生了内存消耗的问题
```

练习1

```
 /* 这里就是声明了一个全局变量buttons而已，不要去想页面按钮什么的*/
var buttons = [{ name: 'b1' }, { name: 'b2' }, { name: 'b3' }];
function bind() {
//i是3的时候，  3 < buttons.length 是false， 循环结束， i就是3了
    for (var i = 0; i < buttons.length; i++) {

//1、只不过就是给每个元素追加一个新的属性onclick;
//2、不是什么点击事件
        buttons[i].onclick = (function (n) {
            return function () {
            console.log(n);
           }
      })(i);
    }
};
```

```
    bind();//i定格在3
    console.log(buttons);
    buttons[0].onclick();//0
    buttons[1].onclick();//1
    buttons[2].onclick();//2
```

练习2

```
/**
1、声明了一个全局的函数 fun;
2、该函数有两个形参n和o;
3、函数的第一行代码是打印第二个参数 o，只有这里在打印;
4、函数的返回值是一个对象:
   a)有个属性'fun'，里面放的是函数;
   b)这里的fun是字面量;
*/

function fun(n, o) {
     console.log(o);//只有这里在打印
         return {
         fun: function (m) {
              return fun(m, n);
           }
       };
  }

  //var a = fun(0); //undefined
  /** var a = fun(0); 解析:
      0、AO: {n:0, o:undefined}
      1、在调用函数的时候，如果没有给某个形参传值，那么该形参的值就是undefined;
      2、0 传给了形参n，o啥也没收到;
      3、console.log(o); 就打印undefined;
      4、a 缓存的就是函数返回值 {
           fun: function (m) {
               return fun(m, n);
            }
         };
  */
       //a.fun(1);//0
```

```
        /**
        a.fun(1);解析：
          0、a.fun 拿到的是 function (m) {
                                  return fun(m, n);
                                }
          1、a.fun(1)进行函数调用:1赋值给形参m，  return fun(m:1, n:0);
          2、调用的fun函数在GO里面找到了，也就是全局函数fun(1, 0);
          3、实参1赋值给n,实参0赋值给o;
          4、打印的就是第二个形参o，所以打印0;
         */

        //a.fun(2);//0  道理同上，因没有对a进行重新赋值，所以a指向的空间没有发生变化
        //a.fun(3);//0  没有对a进行重新赋值，所以a指向的空间里面的n一直是0

function fun(n, o) {
    console.log(o);//只有这里在打印
        return {
        fun: function (m) {
            return fun(m, n);
          }
      };
  }
        //var b = fun(0).fun(1).fun(2).fun(3).fun(100).fun(1);//undefined 0 1 2 3 100 1
        /** 每调用一次fun，就会更新一次b的值，所以对应的空间里面的n一直在变
           第一步  undefined: fun(0)
            0、AO: {n:0, o:undefined}
            1、在调用函数的时候，如果没有给某个形参传值，那么该形参的值就是undefined;
            2、0 传给了形参n，o啥也没收到;
            3、console.log(o); 就打印undefined;
            4、fun(0) 返回 {
                fun: function (m) {
                    return fun(m:1, n:0);
                }
            };

            第二步  0: fun(0).fun(1)
            1、AO: {n:1, o:0}
            2、1 传给了形参n，0传给了形参o;
            3、console.log(o); 就打印0;
            4、fun(1) 执行完成后返回 {
```

```
                fun: function (m) {
                    return fun(m, n:0);
                }
            };
            结果n变成m接收到的1

            第三步  1：fun(0).fun(1).fun(2)
            1、AO：{n:2, o:1}
            2、2 传给了形参n，1传给了形参o；
            3、console.log(o); 就打印1；
            4、fun(1) 执行完成后返回 {
                fun: function (m) {
                    return fun(m:2, n:1);
                }
            };
            结果n变成m接收到的2

            第四步  2：fun(0).fun(1).fun(2).fun(3)
            1、AO：{n:3, o:2}
            2、3 传给了形参n，2传给了形参o；
            3、console.log(o); 就打印2；
            4、fun(1) 执行完成后返回 {
                fun: function (m) {
                    return fun(m:3, n:2);
                }
            };
            结果n变成m接收到的3
        */

function fun(n, o) {
    console.log(o);//只有这里在打印
        return {
        fun: function (m) {
            return fun(m, n);
          }
      };
  }
        var c = fun(0).fun(1); //undefined  0
        c.fun(2); //1
```

```
        c.fun(22); //1
        c.fun(2222); //1
        c.fun(244444); //1
        //对c进行修改就会改变
        c = c.fun(3); //1
        c.fun(300)
        c = c.fun(3078890)
        c.fun(0);
```

练习3

```
console.log(sum(2, 3));
console.log(sum(2)(3));
       function  sum(n, m){
           //要判断第二个参数有没有
           if(m === undefined){
               return function(j){
                   return n + j;
               }
           }
           //这里不要写else
           return n + m;
       }
```

练习4:

```
sum(1)(2) // 3
sum(1, 2, 3)(10) // 16
sum(1)(2)(3)(4)(5) // 15

function sum () {
     //将argument转换成数组
     var arr = Array.prototype.slice.call(arguments);
     var fn = function () {
     //拼接多次调用的参数为数组
         var arr1 = Array.prototype.slice.call(arguments);
         //递归调用sum
         return sum.apply(null, arr.concat(arr1));
     }
```

```
      //最后一次返回fn时，自动调用valueOf
       fn.valueOf = function () {
          return arr.reduce(function(a, b) {
              return a + b;
           })
       }
       return fn;
   }
```

函数作为参数被传递：

```
var max=10,
    fn = function (n){
         if (n > max){
      console.log(n) ;
    }
} ;
(function (f){
    var max = 100;
    f(15);
})(fn);
//15
//fn函数作为一个参数被传递进入另一个函数，赋值给f参数。执行f(15)时， max变量的取值是10，而不是100
```

从【自由变量】到【作用域链】：

```
var a = 10;

    function fn() {
        var b = 20;
        console.log(a + b); //这里的a在这里就是一个自由变量
    }
 //在调用fn()函数时，函数体中第6行。取b的值就直接可以在fn作用域中取，因为b就是在这里定义的。
 //而取x的值时，就需要到另一个作用域中取
```

```
var n = 10;
function fn() {
```

```
    console.log(n);
  }

function foo(f) {
    var x = 20;
    (function () {
      f();  // 10，  而不是20
      })();
  }
    foo(fn);
  //要到创建这个函数的那个作用域中取值——是“创建”，而不是“调用”
```

上面描述的只是跨一步作用域去寻找。

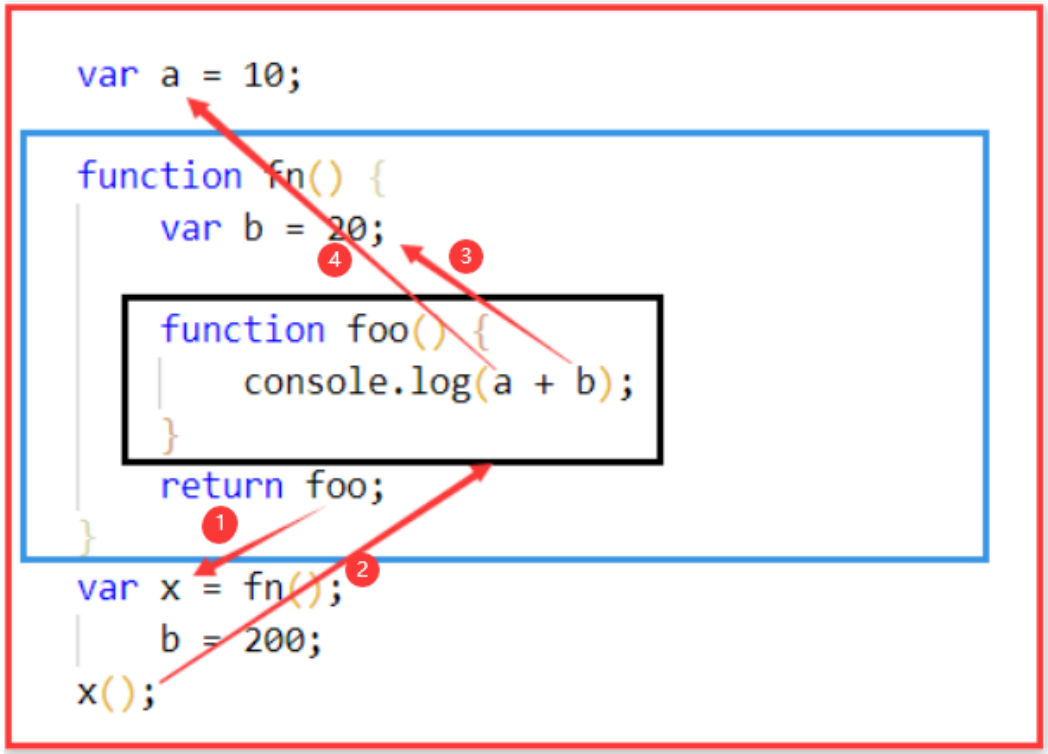
如果跨了一步，还没找到呢？——接着跨！——一直跨到全局作用域为止。要是在全局作用域中都没有找到，那就是真的没有了。

这个一步一步“跨”的路线，我们称之为——作用域链。

获取自由变量时的这个“作用域链”过程：（假设a是自由变量）

- 第一步，现在当前作用域查找a，如果有则获取并结束。如果没有则继续；
- 第二步，如果当前作用域是全局作用域，则证明a未定义，结束；否则继续；
- 第三步，（不是全局作用域，那就是函数作用域）将创建该函数的作用域作为当前作用域；
- 第四步，跳转到第一步。

```
var a = 10;
function fn() {
    var b = 20;
        function foo() {
           console.log(a + b);
      }
    return foo;
  }
  var x = fn();
      b = 200;
  x();  //30
```



x=fn()，fn()返回的是foo函数，赋值给x。执行x()，即执行foo函数代码。取b的值时，直接在fn作用域取出。取a的值时，试图在fn作用域取，但是取不到，只能转向创建fn的那个作用域中去查找，结果找到了。

扩展运算符定义：

扩展运算符(...)是ES6的语法，用于取出参数对象的所有可遍历属性，然后拷贝到当前对象之中。

基本用法:

```
let person = {name: "Amy", age: 15}
let someone = { ...person }
    someone  // {name: "Amy", age: 15}
```

特殊用法:

数组:

由于数组是特殊的对象，所以对象的扩展运算符也可以用于数组。

```
let foo = { ...['a', 'b', 'c'] };
    foo
// {0: "a", 1: "b", 2: "c"}
```

空对象:

如果扩展运算符后面是一个空对象，则没有任何效果。

```
let a = {...{}, a: 1}
```

```
    a // { a: 1 }
```

Int类型、Boolen类型、undefined、null
如果扩展运算符后面是上面这几种类型，都会返回一个空对象，因为它们没有自身属性。

```
// 等同于 {...Object(1)}
{...1} // {}

// 等同于 {...Object(true)}
{...true} // {}

// 等同于 {...Object(undefined)}
{...undefined} // {}

// 等同于 {...Object(null)}
{...null} // {}
```

字符串:
如果扩展运算符后面是字符串，它会自动转成一个类似数组的对象

```
{...'hello'}
// {0: "h", 1: "e", 2: "l", 3: "l", 4: "o"}
```

对象的合并:

```
let age = {age: 15}
let name = {name: "Amy"}
let person = {...age, ...name}
    person; // {age: 15, name: "Amy"}
```

注意事项:
自定义的属性和拓展运算符对象里面属性的相同的时候：
自定义的属性在拓展运算符后面，则拓展运算符对象内部同名的属性将被覆盖掉。

```
let person = {name: "Amy", age: 15};
let someone = { ...person, name: "Mike", age: 17};
    someone;  //{name: "Mike", age: 17}
```

自定义的属性在拓展运算度前面，则变成设置新对象默认属性值。

```
let person = {name: "Amy", age: 15};
let someone = {name: "Mike", age: 17, ...person};
    someone;  //{name: "Amy", age: 15}
```